

3.3 Different ways to add GSAP to a WordPress site

If you want to use **GSAP** in WordPress—especially together with **BricksBuilder**—there are several valid ways to include it. Which one is “best” depends on **how comfortable you are with code**, **how reusable your setup should be**, and **how much control you want**. ☐

This topic is important because many animation problems are *not* caused by GSAP itself, but by **how it was loaded**, **when it becomes available**, or **where your custom code runs**.

In practice, there are **four main approaches** you should know:

1. adding GSAP via a CDN directly in the page or template
2. adding GSAP globally via WordPress enqueueing
3. adding GSAP through a code snippets/plugin-based workflow
4. bundling GSAP in a custom theme or child theme workflow

For a WordPress + BricksBuilder user, you do **not** need to master all of them immediately. But you *should* understand the pros and cons of each, because later—when your projects grow—you may want to switch from a quick setup to a cleaner, more scalable one.

before we compare the methods: what does “adding GSAP” actually mean?

When you “add GSAP” to a website, you usually need to load:

1. the **core GSAP library**
2. optionally one or more **plugins**, such as:
 1. `ScrollTrigger`
 2. `ScrollToPlugin`
 3. `TextPlugin`
4. others depending on your needs

3. your **own custom animation code**, which uses GSAP

A simple mental model is:

1. first the browser loads GSAP
2. then it loads any GSAP plugins
3. then it loads your own script that says what should animate

If this order is wrong, your code may fail.

For example, this will only work if GSAP has already been loaded:

```
gsap.to(".box", {
  x: 200,
  duration: 1
});
```

And this will only work if **both** GSAP and `ScrollTrigger` are already loaded and registered:

```
gsap.registerPlugin(ScrollTrigger);

gsap.to(".box", {
  scrollTrigger: ".box",
  y: 100,
  duration: 1
});
```

So whenever we discuss “ways to add GSAP,” we are really discussing:

1. **where the files come from**
2. **where they are inserted into WordPress**
3. **in what order they are loaded**
4. **whether they are loaded on one page or the whole site**
5. **how easy it is to maintain later**

method 1: adding GSAP via CDN directly in a page, template, or

code block

This is usually the **fastest** way to get started. ☐☐

A **CDN** is a hosted URL from which the browser loads a JavaScript file. Instead of storing GSAP inside your theme, you link to it from an external source.

how this works

You place a `<script>` tag somewhere in your site output, for example:

1. in a custom code area in Bricks
2. in the page/template footer
3. in a header/footer injection plugin
4. in a custom HTML/code element

Example:

```
<script src="https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js"></script>
<script>
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
</script>
```

If you need `ScrollTrigger`, you also load that:

```
<script src="https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/gsap@3/dist/ScrollTrigger.min.js"></script>
<script>
  gsap.registerPlugin(ScrollTrigger);

  gsap.to(".box", {
    scrollTrigger: ".box",
    y: 100,
    duration: 1
  });
</script>
```

where you might do this in WordPress / Bricks

Possible places include:

1. **Bricks page settings** or template-level custom code
2. a **Code element** in Bricks
3. a plugin that allows **header/footer scripts**
4. theme options or site settings that allow custom scripts

why beginners often like this method

Because it is:

1. **quick**
2. **visual**
3. **easy to test**
4. ideal for learning

You can create a section in Bricks, give it a class like `.box`, paste in a script, and immediately see something move.

advantages

1. **very easy to start**
 1. no theme file editing required
 2. no enqueue function needed
 3. minimal WordPress knowledge needed
2. **good for experiments**
 1. useful when learning GSAP basics
 2. useful when building a one-off prototype
 3. useful when testing a small effect on a single page
3. **page-specific loading**
 1. you can load GSAP only where needed
 2. this can be helpful if only one landing page uses animation

disadvantages

1. **harder to maintain**
 1. scripts may end up scattered across pages, templates, and code blocks

2. later you may forget where a certain animation was added
2. **dependency/order issues**
 1. if your custom script runs before GSAP is loaded, it breaks
 2. if `ScrollTrigger` is loaded after your code, it breaks
3. **not ideal for larger projects**
 1. many pages with animation become messy
 2. updating versions across many pages becomes annoying
4. **possible duplication**
 1. you might accidentally load GSAP multiple times
 2. that can lead to confusion or performance issues

when to use it

Use this method if:

1. you are **just learning**
2. you want to test a **small animation quickly**
3. you are building a **simple one-page effect**
4. you are not yet ready to work with theme files or enqueue functions

when not to use it

Avoid relying on this as your long-term default if:

1. your site has many animated pages
2. you want a clean, scalable setup
3. you work on client projects that need easier maintenance
4. you want centralized control over scripts

practical recommendation

For a beginner in BricksBuilder, this is an excellent **sandbox method**. Learn here first. But do not assume it is always the best production architecture.

method 2: adding GSAP properly via WordPress enqueueing

This is the **WordPress-native** way, and in many cases the **best long-term solution** ☐

WordPress has a built-in system for loading CSS and JavaScript files called **enqueueing**. Instead of manually printing `<script>` tags wherever you want, you tell WordPress which scripts to load, and WordPress handles output and ordering more cleanly.

why enqueueing matters

When scripts are enqueued properly, WordPress can better manage:

1. **dependencies**
2. **load order**
3. **footer vs header loading**
4. **conditional loading**
5. **conflicts with themes/plugins**

This makes your setup more robust.

a basic example

You would typically add this to your theme's `functions.php` file, or better, to a **child theme**:

```
function mytheme_enqueue_gsap() {
    wp_enqueue_script(
        'gsap',
        'https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js',
        array(),
        '3.12.5',
        true
    );

    wp_enqueue_script(
        'gsap-scrolltrigger',
        'https://cdn.jsdelivr.net/npm/gsap@3/dist/ScrollTrigger.min.js',
        array('gsap'),
        '3.12.5',
        true
    );

    wp_enqueue_script(
        'my-gsap-animations',
```

```
        get_stylesheet_directory_uri() . '/js/animations.js',
        array('gsap', 'gsap-scrolltrigger'),
        '1.0',
        true
    );
}
add_action('wp_enqueue_scripts', 'mytheme_enqueue_gsap');
```

Your custom animation code would then live in a separate file, for example `/js/animations.js`:

```
gsap.registerPlugin(ScrollTrigger);

gsap.to(".box", {
    scrollTrigger: ".box",
    y: 100,
    duration: 1
});
```

what this code means

Let's unpack it carefully:

1. `wp_enqueue_script(...)`
 1. tells WordPress to load a JavaScript file
2. `'gsap'`
 1. this is the script handle
 2. WordPress uses it as an internal name
3. `'https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js'`
 1. this is the file URL
4. `array()`
 1. this means GSAP core has no dependency in this setup
5. `'3.12.5'`
 1. this is the version string
 2. useful for cache-busting and clarity
6. `true`
 1. load in the footer
 2. generally good for performance and safer for DOM-related code

For `ScrollTrigger`, this line matters a lot:

```
array('gsap')
```

That tells WordPress:

```
“Load ScrollTrigger only after gsap has loaded.”
```

And here:

```
array('gsap', 'gsap-scrolltrigger')
```

you tell WordPress:

```
“Load my animation file only after both GSAP core and ScrollTrigger are available.”
```

That is one of the biggest advantages of enqueueing.

advantages

1. **cleaner structure**
 1. GSAP loading is centralized
 2. your animation logic lives in its own file
2. **better dependency management**
 1. WordPress handles load order
 2. fewer “gsap is not defined” errors
3. **more scalable**
 1. better for larger sites
 2. better for reusable workflows
 3. better for client projects
4. **easier maintenance**
 1. update a version in one place
 2. update one JS file instead of many inline snippets
5. **closer to professional practice**
 1. this is the way many developers structure scripts in WordPress

disadvantages

1. **requires some PHP**
 1. even if only a little
 2. this may feel intimidating at first
2. **slower to set up initially**

1. compared to just pasting script tags into a page
3. **you need file access**
 1. child theme
 2. theme editor
 3. SFTP or hosting file manager
 4. a code snippets workflow that supports this logic

when to use it

This is ideal if:

1. you want a **solid long-term setup**
2. you animate across **multiple pages**
3. you want your code to stay organized
4. you are ready to learn a little WordPress development structure

important best practice: use a child theme

If you place code directly in a parent theme and later update the theme, your changes may be overwritten. ☐

So if you are editing `functions.php`, it is usually safer to do so in a **child theme**.

conditional loading

A very useful improvement is to load GSAP only on pages that actually need it.

Example:

```
function mytheme_enqueue_gsap() {
    if (!is_page('landing-page')) {
        return;
    }

    wp_enqueue_script(
        'gsap',
        'https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js',
        array(),
        '3.12.5',
        true
    );
}
```

```
wp_enqueue_script(
    'my-gsap-animations',
    get_stylesheet_directory_uri() . '/js/landing-page-animations.js',
    array('gsap'),
    '1.0',
    true
);
}
add_action('wp_enqueue_scripts', 'mytheme_enqueue_gsap');
```

This means:

1. if the current page is **not** `landing-page`, stop immediately
2. otherwise, load GSAP and your animation file

This is cleaner than loading everything site-wide when only one page needs it.

method 3: adding GSAP through a code snippets plugin or script management plugin

This is a kind of **middle ground** between “quick and easy” and “properly structured.” ☘

Instead of editing theme files manually, you use a plugin that lets you insert PHP, JavaScript, or site-wide scripts in a managed interface.

Examples of plugin categories:

1. **Code Snippets** plugins
2. **Header/Footer script** plugins
3. **asset/script manager** plugins
4. some advanced **custom code manager** plugins

how this works

There are two main variations:

1. you insert raw `<script>` tags globally
2. you add PHP that uses `wp_enqueue_script()`

The second is generally better if the plugin supports PHP snippets safely.

example: enqueueing GSAP through a snippets plugin

```
function mysite_enqueue_gsap() {
    wp_enqueue_script(
        'gsap',
        'https://cdn.jsdelivr.net/npm/gsap@3/dist/gsap.min.js',
        array(),
        '3.12.5',
        true
    );

    wp_enqueue_script(
        'gsap-scrolltrigger',
        'https://cdn.jsdelivr.net/npm/gsap@3/dist/ScrollTrigger.min.js',
        array('gsap'),
        '3.12.5',
        true
    );
}

add_action('wp_enqueue_scripts', 'mysite_enqueue_gsap');
```

Then you might place your custom JS:

1. in a separate JS file
2. in a page-level code area
3. in another snippet/plugin field depending on the plugin

why this is attractive

For many WordPress users, especially non-developers, this is easier because:

1. no theme file editing is required
2. your custom code survives theme updates

3. plugin interfaces can be less intimidating than file editing

advantages

1. **safer than editing parent theme files**
 1. theme updates are less risky
2. **often easier for non-developers**
 1. no SFTP required
 2. no direct file management required
3. **can still be fairly organized**
 1. especially if the plugin supports named snippets
 2. you can label snippets clearly
4. **good transition step**
 1. excellent if you are learning WordPress development concepts gradually

disadvantages

1. **depends on another plugin**
 1. more plugin dependency
 2. plugin settings differ from tool to tool
2. **can still become messy**
 1. if you store too many unrelated scripts in too many snippets
3. **sometimes less transparent**
 1. another developer may need to search plugins to find your script setup
4. **not always ideal for larger development workflows**
 1. especially if you later move toward version-controlled theme/project files

when to use it

This is a strong option if:

1. you don't want to edit theme files yet
2. you want something more maintainable than random page-level script tags
3. you are comfortable managing code in a plugin interface
4. you want a relatively safe setup on client sites

a very practical beginner-friendly approach

A nice compromise can be:

1. enqueue **GSAP and plugins globally or conditionally** via a snippets plugin
2. keep **small page-specific animation code** inside Bricks where it is used
3. move larger animation code into separate JS files later

That way, your **library loading** is centralized, but your **small experiments** remain easy to work with.

method 4: bundling GSAP in a custom theme or child theme workflow

This is the most “developer-style” approach and often the cleanest for serious projects ☐

Instead of relying on CDN links everywhere, you may keep your scripts in your project structure and load them through your theme or build process.

what this can mean

This can range from relatively simple to quite advanced:

1. storing your own custom JS files inside the theme
2. enqueueing local copies of GSAP files
3. using a build tool or bundler
4. organizing animations into modules/files

For your current level, you probably do **not** need the advanced version yet. But it is good to understand the direction.

local file example

Suppose you store GSAP files inside your child theme:

1. `/js/vendor/gsap.min.js`
2. `/js/vendor/ScrollTrigger.min.js`
3. `/js/animations.js`

Then enqueue them:

```
function mytheme_enqueue_local_gsap() {
    wp_enqueue_script(
        'gsap',
        get_stylesheet_directory_uri() . '/js/vendor/gsap.min.js',
        array(),
        '3.12.5',
        true
    );

    wp_enqueue_script(
        'gsap-scrolltrigger',
        get_stylesheet_directory_uri() . '/js/vendor/ScrollTrigger.min.js',
        array('gsap'),
        '3.12.5',
        true
    );

    wp_enqueue_script(
        'my-animations',
        get_stylesheet_directory_uri() . '/js/animations.js',
        array('gsap', 'gsap-scrolltrigger'),
        '1.0',
        true
    );
}

add_action('wp_enqueue_scripts', 'mytheme_enqueue_local_gsap');
```

CDN vs local file: what is the difference?

With a CDN:

1. files are loaded from an external server

With local files:

1. files are served from your own site/theme

Neither is automatically “always better.” It depends on priorities.

CDN advantages

1. quick to set up
2. easy to update URL versions
3. no need to manage local vendor files manually

local file advantages

1. more project control
2. less dependence on third-party external URLs
3. easier to version together with the project files
4. often preferable in structured development workflows

disadvantages of a custom theme workflow

1. more setup complexity
2. requires more confidence with file structure
3. may involve build tooling later
4. can feel like overkill for small sites

when to use it

Use this if:

1. you build custom WordPress sites regularly
 2. you want a reusable professional workflow
 3. you manage code in a structured way
 4. you may later adopt Git/version control and more advanced JS organization
-

method 5: loading GSAP only where Bricks needs it

This is not a totally separate technical method, but rather an important **strategy** for WordPress + BricksBuilder projects.

A common beginner mistake is:

1. load GSAP everywhere
2. put all animation logic in one giant script
3. target many classes globally
4. lose track of what affects what

A better approach is often:

1. load the library globally *or conditionally*
2. keep animation code grouped by template/page/component
3. use consistent classes and naming

For example:

1. one JS file for hero animations
2. one JS file for scroll sections
3. one JS file for reusable components like sliders/cards
4. or one main file with clearly separated sections

This becomes especially useful in Bricks because Bricks encourages visual building, and without structure your JS can become hard to map back to the layout.

a realistic comparison of the methods

Here is the practical comparison in plain language:

1. **CDN directly in page/template**
 1. best for: learning, testing, small one-off effects
 2. weakest point: maintenance becomes messy
2. **WordPress enqueueing in theme/child theme**
 1. best for: long-term clean setup
 2. weakest point: requires some PHP confidence
3. **code snippets/plugin-based loading**
 1. best for: users who want structure without editing theme files
 2. weakest point: still plugin-dependent and can become messy if overused
4. **custom theme/local asset workflow**
 1. best for: advanced or growing professional projects
 2. weakest point: more setup complexity

If I were advising someone with **limited JS/CSS skills** but who wants to learn properly, I would suggest this progression:

1. start with **page-level CDN loading** to understand GSAP basics
2. then move to **snippet/plugin-based enqueueing**
3. then learn **child theme enqueueing**
4. later, if needed, adopt a more structured local/bundled workflow

That learning path keeps your cognitive load manageable. ☐

important technical details you should understand regardless of method

No matter which loading method you choose, a few concepts always matter.

1. load order

Your custom GSAP code must run **after** GSAP is available.

Bad order:

1. your animation code runs
2. GSAP loads afterward

Result: error such as `gsap is not defined`

Good order:

1. GSAP core loads
 2. GSAP plugin loads
 3. your custom animation code runs
-

2. plugin registration

Some GSAP plugins should be registered before use:

```
gsap.registerPlugin(ScrollTrigger);
```

If you forget this, your scroll-based animations may not work correctly.

3. DOM readiness

Even if GSAP is loaded correctly, the elements you want to animate must also exist when your code runs.

For example, this may fail if `.box` is not yet in the DOM at execution time:

```
gsap.to(".box", {  
  x: 200  
});
```

A safer beginner pattern is:

```
document.addEventListener("DOMContentLoaded", function () {  
  gsap.to(".box", {  
    x: 200,  
    duration: 1  
  });  
});
```

This says:

1. wait until the HTML document has loaded
2. then run the animation code

This is especially useful in WordPress because page builders, templates, and dynamic content can make script timing less obvious.

4. page builders can change markup

With BricksBuilder, what you see visually is tied to generated HTML. If you later change:

1. a class name
2. a nested structure
3. a template assignment

your GSAP selector may stop matching.

So this selector:

```
gsap.to(".hero-title", {  
  y: 50  
});
```

only works if an element with class `.hero-title` still exists on the front end.

That is why consistent class naming is very important.

5. avoid loading the same library multiple times

This is a surprisingly common issue.

For example:

1. GSAP loaded in a header/footer plugin
2. GSAP loaded again in Bricks custom code
3. GSAP loaded a third time in `functions.php`

This can create debugging confusion. Even if it does not always break visibly, it is bad practice.

Try to keep a clear answer to the question:

“Where exactly is GSAP being loaded on this site?”

If you cannot answer that in one sentence, your setup may already be getting messy.

recommended setups for different skill levels

if you are an absolute beginner

Use:

1. **CDN + page-level script**
2. maybe inside a Bricks code block or page custom code area

Why:

1. fastest feedback loop
2. easiest for learning syntax
3. minimal setup complexity

But keep it limited to experiments.

if you are a beginner who wants a cleaner real-site setup

Use:

1. a **Code Snippets**-style plugin to enqueue GSAP
2. a separate custom JS file if possible
3. small page-specific code only where needed

Why:

1. safer than editing theme files too early
 2. more maintainable than scattered inline scripts
 3. good stepping stone toward proper development practice
-

if you are building more serious projects

Use:

1. **child theme enqueueing**
2. separate JS files
3. conditional loading where sensible
4. clear naming conventions for Bricks elements

Why:

1. cleaner architecture
 2. easier maintenance
 3. better scalability
-

recommended setup specifically for WordPress + BricksBuilder learners

For *your* situation, I would recommend the following staged approach:

1. **phase 1: learn GSAP with the simplest possible setup**
 1. load GSAP via CDN
 2. paste small scripts into a controlled test page in Bricks
 3. animate simple classes like `.box`, `.title`, `.button`
2. **phase 2: centralize library loading**
 1. move GSAP loading into a snippet/plugin-based global setup
 2. stop loading the library individually on every test page
3. **phase 3: separate your animation logic**
 1. move your custom animations into a JS file
 2. keep the file organized by section or component
 3. use comments to mark each animation area
4. **phase 4: move to child theme enqueueing when you feel ready**
 1. this gives you a more professional workflow
 2. especially useful for client work or bigger sites

This sequence is effective because it respects the idea that learning has a kind of “cost function” ☐☐

If we think of setup difficulty as D and learning benefit as B , then for beginners you want a method where the ratio $\frac{B}{D}$ is as high as possible. Early on, simple setups often give better returns; later, cleaner architecture becomes more valuable.

a simple decision guide

If you are unsure which method to choose, use this rule of thumb:

1. **I just want to test GSAP today**

1. use CDN in the page/template
 2. **I want a safer no-theme-edit workflow**
 1. use a snippets/plugin approach
 3. **I want a clean long-term WordPress setup**
 1. use child theme enqueueing
 4. **I build structured custom projects**
 1. use a custom theme/local asset workflow
-

common mistakes to avoid

1. **putting GSAP code before the library is loaded**
 2. **forgetting to load or register plugins like `ScrollTrigger`**
 3. **animating selectors that do not exist on the front end**
 4. **loading GSAP multiple times from different places**
 5. **storing too much important logic in scattered inline scripts**
 6. **editing a parent theme directly**
 7. **not checking whether scripts should load globally or only on specific pages**
-

my practical recommendation in one sentence

For learning, start with **CDN + simple page-level code**; for real projects, move as soon as possible toward **centralized loading via snippets or child-theme enqueueing**. ☐

Revision #2

Created 2026-04-26 09:52:10 UTC by art10m

Updated 2026-04-26 10:04:21 UTC by art10m